Concolic Execution Tailored for Hybrid Fuzzing

Insu Yun
Georgia Institute of Technology
We are using many software applications
We are using many (vulnerable) software applications
Attack flow (hacking)

Vulnerability
Attack flow (hacking)

Vulnerability

Exploitation
Attack flow (hacking)

Vulnerability

Exploitation

Compromise
Attack flow (hacking)

Vulnerability → Exploitation → Compromise

Automatic vulnerability finding
- APISan (Security ’16), CAB-Fuzz (ATC ‘16),
- QSYM (Security’ 18), DIE(Oakland ‘20),
- Hybridra(ongoing)
Attacks flow (hacking)

Vulnerability → Exploitation → Compromise

Automatic vulnerability finding
- APISan (Security '16)
- QSYM (Security '18)
- Hybridra (ongoing)
- CAB-Fuzz (ATC '16)
- DIE (Oakland '20)

Automatic exploitation
- ArcHeap (Security '20)
Attack flow (hacking)

Vulnerability → Exploitation → Compromise

Automatic vulnerability finding
- APISan (Security ’16), CAB-Fuzz (ATC ’16),
- QSYM (Security’ 18), DIE(Oakland ‘20)
- Hybridra(Ongoing)

Automatic exploitation
- ArcHeap (Security ‘20)

Fixing / Mitigation
- HDFI (Oakland ’16), REPT (OSDI ‘18)
- FFMalloc (Security’ 20)
Attack flow (hacking)

Vulnerability → Exploitation → Compromise

Automatic vulnerability finding
- APISan (Security ’16), CAB-Fuzz (ATC ’16),
- QSYM (Security’ 18), DIE(Oakland ’20)
- Hybridra(Ongoing)

Automatic exploitation
- ArcHeap (Security ‘20)

Fixing / Mitigation
- HDFI (Oakland ’16), REPT (OSDI ‘18)
- FFMalloc (Security’ 20)
Today’s talk

QSYM: A Binary-level Concolic Execution Engine for Hybrid fuzzing

Hybridra: A Hybrid Fuzzer for Kernel File Systems

- Binary
- User applications

- Source code
- File systems
Overview of 40-year-old random testing (fuzzing)

Seeds
Overview of 40-year-old random testing (fuzzing)

Seeds → Fuzzer
Overview of 40-year-old random testing (fuzzing)

Seeds → Fuzzer → Test cases

*
Overview of 40-year-old random testing (fuzzing)
Overview of 40-year-old random testing (fuzzing)

Seeds → Fuzzer → Test cases → Program → Crash
Recent breakthrough: Code coverage feedback
Before code coverage feedback,

```python
x = input()

if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```
Before code coverage feedback,

```python
x = input()
if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```

Seeds

x = 'E4SY'
Before code coverage feedback, 

```python
x = input()
if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```

Seeds

- x = 'E4SY'

Test cases

- x = 'E4SI'
Before code coverage feedback,

```python
x = input()
if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```

Seeds

- x = 'E4SY'

Test cases

- x = 'E4SI'
- x = 'S4SY'
Before code coverage feedback,

```python
x = input()
if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()

Seeds
x = 'E4SY'

Test cases
x = 'E4SI'	x = 'S4SY'	x = 'PTY'
```
Before code coverage feedback,

```python
x = input()
if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```

Seeds

- x = 'E4SY'

Test cases

- x = 'E4SI'
- x = 'S4SY'
- x = 'PTSY'
- x = 'H4SY'
Before code coverage feedback,

```python
x = input()
if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```

Seeds

- `x = 'E4SY'`
- `x = 'E4SI'`
- `x = 'H4SY'`
- `x = 'O4SY'`
- `x = 'S4SY'`
- `x = 'PTSY'`
Before code coverage feedback,

\[ x = \text{input()} \]

\[
\text{if } x[0] == 'H':
  \text{if } x[1] == 'A':
    \text{if } x[2] == 'R':
      \text{if } x[3] == 'D':
        \text{crash()}
\]

**Seeds**

- \( x = 'E4SY' \)
- \( x = 'E4SI' \)
- \( x = 'H4SY' \)
- \( x = 'O4SY' \)
- \( x = 'PTSY' \)

**Test cases**

- \( x = 'E4SY' \)
- \( x = 'S4SY' \)
- \( x = 'O4SY' \)
- \( \ldots \)
Before code coverage feedback,

\[
x = \text{input()}
\]

\[
\begin{align*}
\text{if } x[0] &= 'H': \\
&\quad \text{if } x[1] = 'A': \\
&\quad\quad \text{if } x[2] = 'R': \\
&\quad\quad\quad \text{if } x[3] = 'D': \\
&\quad\quad\quad\quad \text{crash}()
\end{align*}
\]

\[
P(\text{crash}) = 2^{-32}
\]
After code coverage feedback,

```
x = input()

if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```

Seeds

$x = \text{'E4SY'}$
After code coverage feedback,

```python
x = input()
if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```

Seeds

```
x = 'E4SY'
```

Test cases

```
x = 'E4SI'
```
After code coverage feedback,

\[
x = \text{input()}
\]

if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()

Seeds

\[
x = 'E4SY'
\]

Test cases

\[
x = 'E4SI'
\]

\[
x = 'S4SY'
\]
After code coverage feedback,

\[
x = \text{input()}
\]

if \( x[0] == 'H' \):  
  if \( x[1] == 'A' \):  
    if \( x[2] == 'R' \):  
      if \( x[3] == 'D' \):  
        crash()

Seeds

- \( x = 'E4SY' \)
- \( x = 'E4SI' \)
- \( x = 'S4SY' \)
- \( x = 'ETSY' \)

Test cases
After code coverage feedback,

```python
x = input()
if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```

**Seeds**

- `x = 'E4SY'`
- `x = 'H4SY'`
- `x = 'E4SI'`
- `x = 'S4SY'`
- `x = 'ETS4Y'`

**Test cases**

- `x = 'E4SI'`
- `x = 'S4SY'`
- `x = 'ETS4Y'`
- `x = 'H4SY'`
After code coverage feedback,

```
x = input()
if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```

Seeds

- x = 'E4SY'

Test cases

- x = 'E4SI'
- x = 'S4SY'
- x = 'ETSY'
- x = 'H4SY'
After code coverage feedback,

```python
x = input()
if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()

Seeds

x = 'E4SY'

Test cases

x = 'E4SI'
x = 'S4SY'
x = 'ETSY'

x = 'H4SY'

New code coverage!
```
Generate test cases from a test case that introduces new code coverage

```python
x = input()
if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```

Seeds

- x = 'EASY'
- x = 'H4SY'
Generate test cases from a test case that introduces new code coverage

```python
x = input()
if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```

Seeds

- `x = 'EASY'`
- `x = 'H4SY'`

Test cases

- `x = 'H4SI'`
Generate test cases from a test case that introduces new code coverage

```py
x = input()
if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```

Seeds

- x = 'EASY'
- x = 'H4SY'

Test cases

- x = 'H4SI'
- x = 'HASY'

New code coverage!
Generate test cases from a test case that introduces new code coverage

\[
x = \text{input()}
\]
\[
\text{if } x[0] == 'H':
\]
\[
\quad \text{if } x[1] == 'A':
\]
\[
\quad \quad \text{if } x[2] == 'R':
\]
\[
\quad \quad \quad \text{if } x[3] == 'D':
\]
\[
\quad \quad \quad \quad \text{crash()}
\]

Seeds

\[
x = 'EASY'
\]
\[
x = 'H4SY'
\]

Test cases

\[
x = 'H4SI'
\]
\[
x = 'HASY'
\]

New code coverage!

\[
P(\text{crash}) = 2^{-32}
\]
Generate test cases from a test case that introduces new code coverage

```
x = input()
if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```

Seeds

- $x = 'EASY'$
- $x = 'H4SY'$

Test cases

- $x = 'HASI'$
- $x = 'HASY'$

**New code coverage!**

$$P(\text{crash}) = 2^{-32} = 2^{-8} \times 2^{-2} = 2^{-10}$$

Per-byte $4$ bytes
Coverage-guided fuzzing is effective

- Fuzzer developed by Google
- Re-discover coverage-guided fuzzing
- Found hundreds of bugs in many programs e.g., Safari, Firefox, OpenSSL, …
Coverage-guided fuzzing is effective

- Fuzzer developed by Google
- Re-discover coverage-guided fuzzing
- Found hundreds of bugs in many programs e.g., Safari, Firefox, OpenSSL, ...

- LLVM community developed
- A library to include random testing as a part of projects e.g., LLVM, Chromium, Tensorflow, ...
Coverage-guided fuzzing is effective

- Fuzzer developed by Google
- Re-discover coverage-guided fuzzing
- Found hundreds of bugs in many programs e.g., Safari, Firefox, OpenSSL, ...

- LLVM community developed
- A library to include random testing as a part of projects e.g., LLVM, Chromium, Tensorflow, ...

- Use Google’s cloud resources to fuzz open-source software
- 4 trillion test cases a week
Limitations of fuzzing: Randomly hard-to-find

```python
x = int(input())

// 459684 == 678²
if x * x == 459684:
    crash()
```
Limitations of fuzzing: Randomly hard-to-find

\[ x = \text{int}(\text{input}()) \]

// 459684 == 678^2

if \( x \times x = 459684 \):
    \text{crash}()

x = 0

Seeds

x = 0
Limitations of fuzzing: Randomly hard-to-find

\[ x = \text{int(input())} \]

// 459684 == 678^2
if x * x == 459684:
    crash()

Seeds

Test cases

\[ x = 0 \]

\[ x = 3 \]
Limitations of fuzzing: Randomly hard-to-find

```python
x = int(input())

// 459684 == 678^2
if x * x == 459684:
    crash()
```

Seeds

- **x = 0**

Test cases

- **x = 3**
- **x = 452**
Limitations of fuzzing: Randomly hard-to-find

\[
x = \text{int}(\text{input}())
\]

\[
// 459684 == 678^2
\]

\[
\text{if } x \times x == 459684 : \\
\text{crash()}
\]

Seeds

Test cases

\[
x = 0
\]

\[
x = 3
\]

\[
x = 452
\]

\[
x = 942
\]
Limitations of fuzzing: Randomly hard-to-find

```python
x = int(input())

// 459684 == 678^2
if x * x == 459684:
    crash()
```

Seeds

- x = 0

Test cases

- x = 3
- x = 452
- x = 942
- x = 512
Limitations of fuzzing: Randomly hard-to-find

\[ x = \text{int}(\text{input}()) \]

\[
// 459684 == 678^2 \\
\text{if } x \times x == 459684 : \\
\text{crash()} 
\]
Limitations of fuzzing: Randomly hard-to-find

```python
x = int(input())
// 459684 == 678^2
if x * x == 459684:
    crash()
```

Seeds

- x = 0

Test cases

- x = 3
- x = 452
- x = 942
- x = 512
- x = 28
- ...

12
Limitations of fuzzing: Randomly hard-to-find

\[
x = \text{int}(\text{input}())
\]

// 459684 == 678^2
if x * x == 459684 :
  crash()

Seeds

Test cases

P(cris) = 2^{-32}
Limitations of coverage-guided fuzzing

Can we do better?

Fuzzer → Test cases → Program → Crash

Code coverage feedback
Limitations of coverage-guided fuzzing

Seeds $\rightarrow$ Fuzzer $\rightarrow$ Test cases $\rightarrow$ Program $\rightarrow$ Crash

Code coverage feedback
Limitations of coverage-guided fuzzing

Seeds → Fuzzer → Test cases → Program → Crash

Code coverage feedback + Concolic execution
Limitations of coverage-guided fuzzing

Seeds → Fuzzer → Test cases → Program → Crash

Code coverage feedback

Concolic execution

Hybrid fuzzing
Concolic execution can help fuzzing by finding hard-to-find test cases

```plaintext
x = int(input())

// 459684 == 678^2
if x * x == 459684:
    crash()
```
Concolic execution can help fuzzing by finding hard-to-find test cases

```python
x = int(input())

// 459684 == 678^2
if x * x == 459684:
    crash()
```
Concolic execution can help fuzzing by finding hard-to-find test cases

```python
x = int(input())

// 459684 == 678^2
if x * x == 459684 :
    crash()
```

```
x = input()

if x * x == 459684 :
```

145
Concolic execution can help fuzzing by finding hard-to-find test cases

```
x = int(input())

// 459684 == 678^2
if x * x == 459684 :
crash()
```

```
x = input()

if x * x == 459684 :
x = 0
```
Concolic execution can help fuzzing by finding hard-to-find test cases

```python
x = int(input())

// 459684 == 678^2
if x * x == 459684:
    crash()

if x * x == 459684:
    x = 0
```

Constraint solver
Concolic execution can help fuzzing by finding hard-to-find test cases

\[
x = \text{int}(\text{input}())
\]

\[
// 459684 == 678^2
\]

\[
\text{if } x \times x == 459684 : \quad \text{crash()}
\]

\[
x = \text{input}()
\]

\[
\text{if } x \times x == 459684 : \quad \text{Constraint solver}
\]

\[
x = 0 \quad x = 678
\]
Concolic execution can help fuzzing by finding hard-to-find test cases

```
x = int(input())
// 459684 == 678^2
if x * x == 459684:
    crash()
```
Hybrid fuzzing has achieved great success in small-scale study (DARPA Cyber Grand Challenge)
Hybrid fuzzing has achieved great success in small-scale study (DARPA Cyber Grand Challenge)

• Organized by DARPA in 2017
• Build a system to find bugs, exploit and patch on binaries

• Over 100 teams → 7 teams were qualified (include our team)
• Almost $4 million for prize money
Hybrid fuzzing has achieved great success in small-scale study (DARPA Cyber Grand Challenge)

- Organized by DARPA in 2017
- Build a system to find bugs, exploit and patch on binaries

- Over 100 teams → 7 teams were qualified (include our team)
- Almost $4 million for prize money

- Small binaries: a few KB
Hybrid fuzzing has achieved great success in small-scale study (DARPA Cyber Grand Challenge)

- The winner from CMU used hybrid fuzzing

- Shellphish open-sourced their tool, Driller
  - Won 3rd place in CGC competition
  - Found 6 new crashes: cannot be found by fuzzing or concolic execution
But, hybrid fuzzing fails to scale real-world applications

Cannot find ANY bug in real-world software using Driller!
Current concolic executors suffer several problems to be used in hybrid fuzzing

Generating constraints is too slow
Current concolic executors suffer several problems to be used in hybrid fuzzing

- Generating constraints is too slow
- Not effective in generating test cases
Current concolic executors suffer several problems to be used in hybrid fuzzing

- Generating constraints is too slow
- Not effective in generating test cases
Symbolic emulation is well-known to be much slower than concrete execution

```c
int is_double(int* a, int b) {
    return *a == 2 * b;
}
```
Symbolic emulation is well-known to be much slower than concrete execution.

Concrete Execution:  Just Execute!

```c
int is_double(int* a, int b) {
    return *a == 2 * b;
}
```
Symbolic emulation is well-known to be much slower than concrete execution

```c
int is_double(int* a, int b) {
    return *a == 2 * b;
}
```

Symbolic Emulation: Taint tracking
Symbolic emulation is well-known to be much slower than concrete execution

```c
int is_double(int* a, int b) {
    return *a == 2 * b;
}
```

Symbolic Emulation:
- Taint tracking
- Constructing symbols (e.g., *a == 2 * Symbol_b)
Symbolic emulation is well-known to be much slower than concrete execution.

```c
int is_double(int* a, int b) {
    return *a == 2 * b;
}
```

Symbolic Emulation:
- Taint tracking
- Constructing symbols (e.g., *a == 2 * Symbol_b)
- Interpretation (e.g., *a = ???)
Symbolic emulation is well-known to be much slower than concrete execution!

Symbolic Emulation:
- Taint tracking
- Constructing symbols (e.g., *a == 2 * Symbol_b)
- Interpretation (e.g., *a = ???)
State forking can be used to solve this problem
State forking can be used to solve this problem
State forking can be used to solve this problem
State forking can be used to solve this problem
State forking can be used to solve this problem
State forking can be used to solve this problem

No need to re-execute!
State forking is limited in hybrid fuzzing
State forking is limited in hybrid fuzzing

• # of states is enormous in a complex real-world program
  => Large performance overhead

• Hybrid fuzzing explores paths randomly following fuzzing
  => Low reusability
State forking is limited in hybrid fuzzing

• # of states is enormous in a complex real-world program
  => Large performance overhead

• Hybrid fuzzing explores paths randomly following fuzzing
  => Low reusability

State forking cannot help slow symbolic emulation in hybrid fuzzing!
Current concolic executors have several problems to be used in hybrid fuzzing

Generating constraints is too slow

Not effective in generating test cases
Completeness of concolic execution often blocks its further exploration

```
1 // 'buf' and 'x' are symbolic
2 int completeness(char* buf, int x) {
3   very_complicated_logic(buf);
4
5   if (x * x == 1234 * 1234)
6     crash();
7 }
```
Completeness of concolic execution often blocks its further exploration

```
// 'buf' and 'x' are symbolic
int completeness(char* buf, int x) {
    very_complicated_logic(buf);

    if (x * x == 1234 * 1234)
        crash();
}
```
Completeness of concolic execution often blocks its further exploration

```c
// 'buf' and 'x' are symbolic
int completeness(char* buf, int x) {
    very_complicated_logic(buf);
    if (x * x == 1234 * 1234)
        crash();
}
```

Analyze every routine for completeness!
Soundness of concolic execution incurs unnecessary re-execution

```c
// 'x' is symbolic and 'x' == 0 in a given input
int soundness(int x) {
  if (x == 0)
    do_something();
  if (x * x == 1234 * 1234)
    crash();
}
```
Soundness of concolic execution incurs unnecessary re-execution

```c
// 'x' is symbolic and 'x' == 0 in a given input
int soundess(int x) {
    if (x == 0)
        do_something();
    if (x * x == 1234 * 1234)
        crash();
}
```
Soundness of concolic execution incurs unnecessary re-execution

```c
// 'x' is symbolic and 'x' == 0 in a given input
int soundess(int x) {
    if (x == 0) {
        do_something();
    }
    if (x * x == 1234 * 1234) {
        crash();
    }
}
```
Soundness of concolic execution incurs unnecessary re-execution

```c
1 // 'x' is symbolic and 'x' == 0 in a given input
2 int soundess(int x) {
3   if (x == 0) {
4     do_something();
5   }
6   if (x * x == 1234 * 1234) {
7     crash();
8   }
```
Our approach

Generating constraints is too slow

Not effective in generating test cases
Our approach

Generating constraints is too slow

Not effective in generating test cases

Systematic approach for fast symbolic emulation
Our approach

Generating constraints is too slow

Not effective in generating test cases

Systematic approach for fast symbolic emulation

New heuristics for hybrid fuzzing
Our approach: QSYM

Systematic approach for fast symbolic emulation

Instruction-level concolic execution (For binary)

New heuristics for hybrid fuzzing

Optimistic solving and basic block pruning
Our approach: Hybridra

Systematic approach for fast symbolic emulation

New heuristics for hybrid fuzzing

Compilation-based concolic execution (For source code)

Staged reduction + Heuristics from QSYM
Related work: Whitebox fuzzing

• Goal
  • Hybrid fuzzing: Make a test case for fuzzing
  • Whitebox fuzzing: Explore a program state solely

• Exploration
  • Hybrid fuzzing: Random
  • Whitebox fuzzing: Systematic

• Strategy: Hybrid fuzzing’s can be more aggressive thanks to coverage-guided fuzzing (e.g., optimistic solving)
Today’s talk

QSYM: A Binary-level Concolic Execution Engine for Hybrid fuzzing

- Binary
- User applications

Hybridra: A Hybrid Fuzzer for Kernel File Systems

- Source code
- File systems
Our system, QSYM, addresses these issues by introducing several key ideas

- Generating constraints is too slow
- Not effective in generating test cases

- Instruction-level concolic execution (For binary)
- Optimistic solving and basic block pruning
Our system, QSYM, addresses these issues by introducing several key ideas:

- **Generating constraints is too slow**
- **Not effective in generating test cases**

- **Instruction-level concolic execution** (For binary)
- **Optimistic solving and basic block pruning**
QSYM has made several design decisions for improving performance

• Discarding intermediate layer

• Instruction-level symbolic execution
QSYM has made several design decisions for improving performance

• Discarding intermediate layer

• Instruction-level symbolic execution

Simple, but BIG design decision
Hybrid fuzzing in a closer look

Assembly

push ebp
...

Hybrid fuzzing in a closer look

```
push ebp
...
```

Assembly

```
t0 = GET:I32(ebp)
t1 = GET:I32(esp)
t2 = Sub32(t1,0x00000004)
PUT(esp) = t2
STle(t2) = t0
...
```

Intermediate Representation (IR)
Hybrid fuzzing in a closer look

Assembly

push ebp
...

Intermediate Representation (IR)

t0 = GET:I32(ebp)
t1 = GET:I32(esp)
t2 = Sub32(t1,0x00000004)
PUT(esp) = t2
STle(t2) = t0
...

Constraints

a == 0x1337
Λ b * c == 0xc001
...
Λ z == 0xc0de
Hybrid fuzzing in a closer look

Assembly

push ebp
...

Intermediate Representation (IR)

t0 = GET:I32(ebp)
t1 = GET:I32(esp)
t2 = Sub32(t1,0x00000004)
PUT(esp) = t2
STle(t2) = t0
...

Constraints

a == 0x1337
Λ b * c == 0xc001
...
Λ z == 0xc0de

Good: Simplifying implementations
e.g., 981 in x86 vs 115 in VEX
Hybrid fuzzing in a closer look

Assembly

push ebp
...

Intermediate Representation (IR)

t0 = GET:I32(ebp)
t1 = GET:I32(esp)
t2 = Sub32(t1,0x00000004)
PUT(esp) = t2
STle(t2) = t0
...

Constraints

a == 0x1337
Λ b * c == 0xc001
...
Λ z == 0xc0de

Good: Simplifying implementations
e.g., 981 in x86 vs 115 in VEX

Bad: Performance bottleneck
Problems of IR: The number of instructions increase

push ebp
...

Assembly

t0 = GET:I32(ebp)
t1 = GET:I32(esp)
t2 = Sub32(t1,0x00000004)
PUT(esp) = t2
STle(t2) = t0
...

Intermediate Representation (IR)

4.96x increase on average!
Problems of IR: Slow transformation speed

push ebp
...

Intermediate Representation (IR)

\[
\begin{align*}
  t0 &= \text{GET:} \text{I32}(ebp) \\
  t1 &= \text{GET:} \text{I32}(esp) \\
  t2 &= \text{Sub32}(t1, 0x00000004) \\
  &\quad \text{PUT(esp) = t2} \\
  &\quad \text{STle(t2) = t0} \\
  &\quad \ldots
\end{align*}
\]
Problems of IR: Slow transformation speed

Assembly

```
push ebp
...  
```

Intermediate Representation (IR)

```
t0 = GET:I32(ebp)
t1 = GET:I32(esp)
t2 = Sub32(t1,0x00000004)
PUT(esp) = t2
STle(t2) = t0
...
```

Slow...

Cache it!
Side effects of caching: Basic-block granularity
Side effects of caching: Basic-block granularity

• Cache lookup is also slow

• Use basic-block granularity for caching
  • i.e., transform a basic block into IR and cache

• Unfortunately, 30% of instructions in a basic block are symbolic
  → 70% of instructions are executed without need
Side effects of caching: Basic-block granularity

• Cache lookup is also slow

• Use basic-block granularity for caching
  • i.e., transform a basic block into IR and cache

• Unfortunately, 30% of instructions in a basic block are symbolic
  $\rightarrow$ 70% of instructions are executed without need
How to solve this challenge?

Assemble

push ebp
...

Intermediate Representation (IR)

t0 = GET:I32(ebp)
t1 = GET:I32(esp)
t2 = Sub32(t1,0x00000004)
PUT(esp) = t2
STle(t2) = t0
...

Constraints

a == 0x1337
Λ b * c == 0xc001
...
Λ z == 0xc0de

Good: Simplifying implementations
e.g., 981 in x86 vs 115 in VEX

Bad: Performance bottleneck
How to solve this challenge?

Assembly

```
push ebp
...
```

Intermediate Representation (IR)

```
t0 = GET:I32(ebp)
t1 = GET:I32(esp)
t2 = Sub32(t1, 0x00000004)
PUT(esp) = t2
STle(t2) = t0
...
```

Constraints

```
a == 0x1337
Λ b * c == 0xc001
Λ z == 0xc0de
...
```

Good: Simplifying implementations
e.g., 981 in x86 vs 115 in VEX

Bad: Performance bottleneck

Performance is the most important!
This is a non-trivial job (LoC)

4.7K Driller

13K QSYM

https://github.com/angr/angr/tree/master/angr/engines
QSYM reduces the number of symbolically executed instructions

- 126 CGC binaries

2.5x end-to-end performance Improvement
Our system, QSYM, addresses these issues by introducing several key ideas.

- Generating constraints is too slow
- Not effective in generating test cases

- Instruction-level concolic execution (For binary)
- Optimistic solving and basic block pruning
Constraint solving can generate a test case that meets given constraints

\[
\begin{aligned}
a &= 0x1337 \\
\land b \times c &= 0xc001 \\
\ldots \\
\land z &= 0xc0de
\end{aligned}
\]

Constraints
Constraint solving can generate a test case that meets given constraints

a == 0x1337
Λ b * c == 0xc001
...
Λ z == 0xc0de
Constraint solving can generate a test case that meets given constraints

```
a == 0x1337
\Lambda b * c == 0xc001
...
\Lambda z == 0xc0de
```

Constraints

Constraint solver

Test cases
Constraint solving **CANNOT** generate a test case that meets given constraints

\[
\begin{align*}
a &= 0x1337 \\
\land b \times c &= 0xc001 \\
\ldots \\
\land z &= 0xc0de
\end{align*}
\]
Constraint solving **CANNOT** generate a test case that meets given constraints

\[
\begin{align*}
a &= 0x1337 \\
\land b \times c &= 0xc001 \\
\ldots \\
\land z &= 0xc0de
\end{align*}
\]
Constraint solving **CANNOT** generate a test case that meets given constraints

\[
\begin{align*}
a &= 0x1337 \\
\land & \quad b \times c = 0xc001 \\
\cdots \\
\land & \quad z = 0xc0de
\end{align*}
\]

**Constraints**

**NP hard**

**Bad: Cannot get anything from concolic execution**
QSYM solves partial constraints to find some test cases

\[ a = 0x1337 \]
\[ \land b \times c = 0xc001 \]

... 

\[ \land z = 0xc0de \]
QSYM solves partial constraints to find some test cases

\[ a == 0x1337 \]
\[ \Lambda b * c == 0xc001 \]
\[ \ldots \]
\[ \Lambda z == 0xc0de \]

Constraint solver

Test cases

Good: Can get test cases from concolic execution
QSYM solves partial constraints to find some test cases

\[
\begin{align*}
a &= 0x1337 \\
b \cdot c &= 0xc001 \\
\ldots \\
z &= 0xc0de
\end{align*}
\]

Constraint solver

If the test cases are *incorrect*...?

Get test cases from execution

Test cases
In hybrid fuzzing, generating incorrect inputs is fine because of coverage-guided fuzzing.
In hybrid fuzzing, generating incorrect inputs is fine because of coverage-guided fuzzing.

Coverage feedback will *filter out* incorrect test cases!
Optimistic solving helps to find more bugs

• LAVA-M dataset
  • Inject hard-to-reach bugs in real-world applications
Remind: Completeness of concolic execution often blocks its further exploration

```c
// 'buf' and 'x' are symbolic
int completeness(char* buf, int x) {
    very_complicated_logic(buf);

    if (x * x == 1234 * 1234)
        crash();
}
```
Remind: Completeness of concolic execution often blocks its further exploration

```c
// 'buf' and 'x' are symbolic
int completeness(char* buf, int x) {
    very_complicated_logic(buf);
    if (x * x == 1234 * 1234)
        crash();
}
```
Remind: Completeness of concolic execution often blocks its further exploration

```c
// 'buf' and 'x' are symbolic
int completeness(char* buf, int x) {
    very_complicated_logic(buf);
    if (x * x == 1234 * 1234)
        crash();
}
```

Limit symbolic execution for repeatedly executed blocks
Remind: Completeness of concolic execution often blocks its further exploration

```c
// 'buf' and 'x' are symbolic
int completeness(char* buf, int x) {
    very_complicated_logic(buf);
    if (x * x == 1234 * 1234)
        crash();
}
```

Can further explore even with such a complicated routine

Limit symbolic execution for repeatedly executed blocks
Remind: Completeness of concolic execution often blocks its further exploration

```c
// 'buf' and 'x' are symbolic
int completeness(char* buf, int x) {
    very_complicated_logic(buf);

    if (x * x == 1234 * 1234)
        crash();
}
```

Limit symbolic execution for repeatedly executed blocks

Can further explore even with such a complicated routine

Incomplete constraints
Incomplete constraints are not significant in practice for hybrid fuzzing

```java
x = input()
y = input()

// x != 0 is missed because of basic block pruning
```
Incomplete constraints are not significant in practice for hybrid fuzzing

```plaintext
if y == 0xdeadbeef:
x = input()
y = input()
// x != 0 is missed because of basic block pruning
```
Incomplete constraints are not significant in practice for hybrid fuzzing

```plaintext
x = input()
y = input()

// x != 0 is missed because of basic block pruning
```

```plaintext
if y == 0xdeadbeef :
    Independent constraints: Use x != 0 in the input
```
Incomplete constraints are not significant in practice for hybrid fuzzing

x = input()
y = input()

// x != 0 is missed because of basic block pruning

if y == 0xdeadbeef:
    Independent constraints: Use x != 0 in the input

if x == 0xdeadbeef:
Incomplete constraints are not significant in practice for hybrid fuzzing.

```plaintext
x = input()
y = input()

// x != 0 is missed because of basic block pruning
```

```plaintext
if y == 0xdeadbeef :

Independent constraints: Use x != 0 in the input

if x == 0xdeadbeef :

Subsumed constraints
```
Incomplete constraints are not significant in practice for hybrid fuzzing

```plaintext
x = input()
y = input()
// x != 0 is missed because of basic block pruning

if y == Oxdeadbeef:
   Independent constraints: Use x != 0 in the input

if x == Oxdeadbeef:
   Subsumed constraints

if x != Oxdeadbeef:
```
Incomplete constraints are not significant in practice for hybrid fuzzing

```
x = input()
y = input()

// x != 0 is missed because of basic block pruning
```

- **Independent constraints**: Use `x != 0` in the input
  ```
  if y == 0xdeadbeef:
  
  `` `

- **Subsumed constraints**
  ```
  if x == 0xdeadbeef:
  
  `` `

- **Failing**...
  ```
  if x != 0xdeadbeef:
  
  ```
Evaluating QSYM

• Scaling to real-world software?

• How good is QSYM compared to
  • The state-of-art hybrid fuzzing (Driller)
QSYM scales to real-world software

- 13 bugs in real-world software (already tested by fuzzing)

<table>
<thead>
<tr>
<th>Program</th>
<th>CVE</th>
<th>Bug Type</th>
<th>Fuzzer</th>
</tr>
</thead>
<tbody>
<tr>
<td>lepton</td>
<td>CVE-2017-8891</td>
<td>Out-of-bounds read</td>
<td>AFL</td>
</tr>
<tr>
<td>openjpeg</td>
<td>CVE-2017-12878</td>
<td>Heap overflow</td>
<td>OSS-Fuzz</td>
</tr>
<tr>
<td></td>
<td>Fixed by other patch</td>
<td>NULL dereference</td>
<td></td>
</tr>
<tr>
<td>tcpdump</td>
<td>CVE-2017-11543*</td>
<td>Heap overflow</td>
<td>AFL</td>
</tr>
<tr>
<td>file</td>
<td>CVE-2017-1000249*</td>
<td>Stack overflow</td>
<td>OSS-Fuzz</td>
</tr>
<tr>
<td></td>
<td>Wait for patch</td>
<td>NULL dereference</td>
<td>OSS-Fuzz</td>
</tr>
<tr>
<td>libarchive</td>
<td>CVE-2017-6836</td>
<td>Heap overflow</td>
<td>AFL</td>
</tr>
<tr>
<td>audiofile</td>
<td>Wait for patch</td>
<td>Heap overflow \times 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Wait for patch</td>
<td>Memory leak</td>
<td></td>
</tr>
<tr>
<td>ffmpeg</td>
<td>CVE-2017-17081</td>
<td>Out-of-bounds read</td>
<td>OSS-Fuzz</td>
</tr>
<tr>
<td>objdump</td>
<td>CVE-2017-17080</td>
<td>Out-of-bounds read</td>
<td>AFL</td>
</tr>
</tbody>
</table>
QSYM scales to real-world software

- 13 bugs in real-world software (already tested by fuzzing)

<table>
<thead>
<tr>
<th>Program</th>
<th>CVE</th>
<th>Bug Type</th>
<th>Fuzzer</th>
</tr>
</thead>
<tbody>
<tr>
<td>lepton</td>
<td>CVE-2017-8891</td>
<td>Out-of-bounds read</td>
<td>AFL</td>
</tr>
<tr>
<td>openjpeg</td>
<td>CVE-2017-12878</td>
<td>Heap overflow</td>
<td>OSS-Fuzz</td>
</tr>
<tr>
<td>tcpdump</td>
<td>CVE-2017-11543*</td>
<td>NULL dereference</td>
<td>AFL</td>
</tr>
<tr>
<td>file</td>
<td>CVE-2017-1000249*</td>
<td>Heap overflow</td>
<td>OSS-Fuzz</td>
</tr>
<tr>
<td>libarchive</td>
<td>Wait for patch</td>
<td>Stack overflow</td>
<td>OSS-Fuzz</td>
</tr>
<tr>
<td>audiofile</td>
<td>CVE-2017-6836</td>
<td>NULL dereference</td>
<td>AFL</td>
</tr>
<tr>
<td>ffmpeg</td>
<td>CVE-2017-17081</td>
<td>Heap overflow × 3</td>
<td>OSS-Fuzz</td>
</tr>
<tr>
<td>objdump</td>
<td>CVE-2017-17080</td>
<td>Out-of-bounds read</td>
<td>AFL</td>
</tr>
</tbody>
</table>
QSYM scales to real-world software

- 13 bugs in real-world software (already tested by fuzzing)

<table>
<thead>
<tr>
<th>Program</th>
<th>CVE</th>
<th>Bug Type</th>
<th>Fuzzer</th>
</tr>
</thead>
<tbody>
<tr>
<td>lepton</td>
<td>CVE-2017-8891</td>
<td>Out-of-bounds read</td>
<td>AFL</td>
</tr>
<tr>
<td>openjpeg</td>
<td>CVE-2017-12878</td>
<td>Heap overflow</td>
<td>OSS-Fuzz</td>
</tr>
<tr>
<td>tcpcap</td>
<td>CVE-2017-11543*</td>
<td>Heap overflow</td>
<td>AFL</td>
</tr>
<tr>
<td>file</td>
<td>CVE-2017-1000249*</td>
<td>Stack overflow</td>
<td>OSS-Fuzz</td>
</tr>
<tr>
<td>libarchive</td>
<td>Wait for patch</td>
<td>NULL dereference</td>
<td>OSS-Fuzz</td>
</tr>
<tr>
<td>audiofile</td>
<td>CVE-2017-6836</td>
<td>Heap overflow</td>
<td>AFL</td>
</tr>
<tr>
<td>ffmpeg</td>
<td>CVE-2017-17081</td>
<td>Out-of-bounds read</td>
<td>OSS-Fuzz</td>
</tr>
<tr>
<td>objdump</td>
<td>CVE-2017-17080</td>
<td>Out-of-bounds read</td>
<td>AFL</td>
</tr>
</tbody>
</table>

Already heavily fuzzed
QSYM can generate test cases that fuzzing is hard to find

- e.g.) ffmpeg: Not reachable by fuzzing

```c
if( !((ox^(ox+dxw))
    | (ox^(ox+dxh))
    | (ox^(ox+dxw+ dxh))
    | (oy^(oy+dyw))
    | (oy^(oy+dyh))
    | (oy^(oy+dyw+ dyh))) >> (16 + shift)
    && !dxx | dxy | dyx | dyy) & 15
&& !(need_emu&&h>MAX_H || stride > MAX_STRIDE))
{
    // the bug is here ;
}
```
QSYM can generate test cases that fuzzing is hard to find

- e.g.) ffmpeg: Not reachable by fuzzing

```c
if( !((ox ^ (ox + dxw))
     | (ox ^ (ox + dxh))
     | (oy ^ (oy + dyw + dyh))) >> (16 + shift)
    && !((dxx | dxy | dyx | dyy) & 15
      && !(need_emu&&((h>MAX_H || stride > MAX_STRIDE)))

{ // the bug is here ; }
```

Cannot be found by cloud fuzzing
(Note: 4 trillion test cases per week)
QSYM can generate test cases that fuzzing is hard to find

- e.g.) ffmpeg: Not reachable by fuzzing

```cpp
if( !((ox^ox+dxw) | (ox^ox+dyh)) | (ox^ox+dxw+dyh)) >> (16 + shift) && !
(dxx | dxy | dyx | dyy) & 15 && !
(need_emu && (h > MAX_H || stride > MAX_STRIDE)))
{
  // the bug is here
}
```

Cannot be found by cloud fuzzing
(Note: 4 trillion test cases per week)

Found by a **single workstation** using QSYM
QSYM outperforms Driller, the state-of-the-art hybrid fuzzer
QSYM outperforms Driller, the state-of-the-art hybrid fuzzer

- Dataset: 126 CGC binaries
- Compare code coverage achieved by a single run of concolic execution
- QSYM achieved more code coverage in 104 (82%) binaries
QSYM is also practically impactful

- e.g., Rode0day: A monthly competition for automatic bug finding tool

<table>
<thead>
<tr>
<th>Place</th>
<th>Elo score</th>
<th>Team name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1,087</td>
<td>afl-lazy</td>
</tr>
<tr>
<td>2</td>
<td>1,069</td>
<td>itszn</td>
</tr>
<tr>
<td>3</td>
<td>1,027</td>
<td>H3ku</td>
</tr>
<tr>
<td>4</td>
<td>1,017</td>
<td>REDQUEEN</td>
</tr>
<tr>
<td>5</td>
<td>1,062</td>
<td>NU-AFL-QSYM</td>
</tr>
</tbody>
</table>

QSYM is also practically impactful

• e.g., Rode0day: A monthly competition for automatic bug finding tool

Table 2. The overall rankings for top Rode0day competitors after 10 competitions.

<table>
<thead>
<tr>
<th>Place</th>
<th>Elo score</th>
<th>Team name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1,087</td>
<td>afl-lazy</td>
</tr>
<tr>
<td>2</td>
<td>1,069</td>
<td>itszn</td>
</tr>
<tr>
<td>3</td>
<td>1,027</td>
<td>H3ku</td>
</tr>
<tr>
<td>4</td>
<td>1,017</td>
<td>REDQUEEN</td>
</tr>
<tr>
<td>5</td>
<td>1,062</td>
<td>NU-AFL-QSYM</td>
</tr>
</tbody>
</table>

Bugs Found in file2_S2

![Bugs Found in file2_S2](image)

Figure 2. A visualization of when teams found bugs in file2_S2 during the November 2018 Rode0day. Green indicates the bugs found within 24 h of a team’s first score. The stars denote the first team to find a bug.

Today’s talk

QSYM: A Binary-level Concolic Execution Engine for Hybrid fuzzing

• Binary
• User applications

Hybridra: A Hybrid Fuzzer for Kernel File Systems

• Source code
• File systems
Hybridra improves Hydra by supporting concolic image mutation
Hybridra improves Hydra by supporting concolic image mutation

Seeds

Random Mutator (Syscall + Image)

Hydra
Hybridra improves Hydra by supporting concolic image mutation

Seeds → Random Mutator (Syscall + Image) → Test cases
Hybridra improves Hydra by supporting concolic image mutation
Hybridra improves Hydra by supporting concolic image mutation

Seeds $\rightarrow$ Random Mutator (Syscall + Image) $\rightarrow$ Test cases $\rightarrow$ LibOS-based Executor $\rightarrow$ Crash

Hydra

Code coverage feedback
Hybridra improves Hydra by supporting concolic image mutation

Seeds → Random Mutator (Syscall + Image) → Test cases → LibOS-based Executor → Crash

Concolic Mutator (Image)

Code coverage feedback
Hybridra: Key ideas

- Generating constraints is too slow
- Not effective in generating test cases

Compilation-based concolic execution (For source code)

Staged reduction + Heuristics from QSYM
Design: Concolic Image Mutator

Seeds
Design: Concolic Image Mutator

Seeds → LibOS executor for concolic execution
Design: Concolic Image Mutator

Seeds → LibOS executor for concolic execution → Reduced constraints
Design: Concolic Image Mutator

Seeds → LibOS executor for concolic execution → Reduced constraints
Design: Concolic Image Mutator

Seeds → LibOS executor for concolic execution → Reduced constraints → Constraint Solver
Design: Concolic Image Mutator

Seeds → LibOS executor for concolic execution → Reduced constraints → Constraint Solver → Test cases
Design: Concolic Image Mutator

Seeds → LibOS executor for concolic execution → Reduced constraints → Constraint Solver → Test cases
Hybridra utilizes compilation-based concolic execution
Hybridra utilizes compilation-based concolic execution

```c
+ Symbol* symA = getSymbol(a);
+ Symbol* symB = getSymbol(b);
+ Symbol* symC = addSymbol(symA, symB);

int c = a + b;

+ Symbol* symD = getSymbol(d);
+ // Make test cases
+ checkEqual(symC, symD);

if (c == d) {
    ...
```
Hybridra utilizes compilation-based concolic execution

```cpp
+ Symbol* symA = getSymbol(a);
+ Symbol* symB = getSymbol(b);
+ Symbol* symC = addSymbol(symA, symB);

int c = a + b;

+ Symbol* symD = getSymbol(d);
+ // Make test cases
+ checkEqual(symC, symD);

if (c == d) {
    ...
```
Hybridra utilizes compilation-based concolic execution

```c
+ Symbol* symA = getSymbol(a);
+ Symbol* symB = getSymbol(b);
+ Symbol* symC = addSymbol(symA, symB);

int c = a + b;

+ Symbol* symD = getSymbol(d);
+ // Make test cases
+ checkEqual(symC, symD);
```

200x performance improvement compared to QSYM
(NOTE: code is required)
Hybridra’s compilation-based concolic execution, Kirenenko, is useful by supporting multi-threading.

<table>
<thead>
<tr>
<th></th>
<th>CUTE</th>
<th>SymCC</th>
<th>Kirenenko</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>C</td>
<td>LLVM IR</td>
<td>LLVM IR</td>
</tr>
<tr>
<td>Memory modeling</td>
<td>Page table</td>
<td>Page table</td>
<td>Shadow memory</td>
</tr>
<tr>
<td>Multi-threading</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
Hybridra’s compilation-based concolic execution, Kirenenko, is useful by supporting multi-threading.

<table>
<thead>
<tr>
<th></th>
<th>CUTE</th>
<th>SymCC</th>
<th>Kirenenko</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>C</td>
<td>LLVM IR</td>
<td>LLVM IR</td>
</tr>
<tr>
<td>Memory modeling</td>
<td>Page table</td>
<td>Page table</td>
<td>Shadow memory</td>
</tr>
<tr>
<td>Multi-threading</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
Hybridra’s compilation-based concolic execution, Kirenenko, is useful by supporting multi-threading.

<table>
<thead>
<tr>
<th></th>
<th>CUTE</th>
<th>SymCC</th>
<th>Kirenenko</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>C</td>
<td>LLVM IR</td>
<td>LLVM IR</td>
</tr>
<tr>
<td>Memory modeling</td>
<td>Page table</td>
<td>Page table</td>
<td>Shadow memory</td>
</tr>
<tr>
<td>Multi-threading</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>File system</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>btrfs</td>
<td>z3 exception</td>
</tr>
<tr>
<td>ext4</td>
<td>Deadlock</td>
</tr>
<tr>
<td>f2fs</td>
<td>Deadlock</td>
</tr>
<tr>
<td>xfs</td>
<td>Deadlock</td>
</tr>
</tbody>
</table>
Hybridra’s compilation-based concolic execution, Kirenenko, is useful by supporting multi-threading

<table>
<thead>
<tr>
<th>Language</th>
<th>CUTE</th>
<th>SymCC</th>
<th>Kirenenko</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory modeling</td>
<td>Page table</td>
<td>LLVM IR</td>
<td>LLVM IR</td>
</tr>
<tr>
<td>Multi-threading</td>
<td>Page table</td>
<td>Page table</td>
<td>Shadow memory ✔</td>
</tr>
</tbody>
</table>
Comparison: Memory modeling

SymCC

Kirenenko

Memory

Shadow memory
Comparison: Memory modeling

Thread1

SymCC

Memory

Shadow memory

Kirenenko
Comparison: Memory modeling

Thread1

SymCC

Thread2

Memory

Shadow memory

Kirenenko
Comparison: Memory modeling

SymCC

Thread1

Thread2

Kirenenko

Memory

Shadow memory
Comparison: Memory modeling

Kirenenko successfully performs concolic execution on file systems in library OS!
Design: Concolic Image Mutator

- Seeds
- LibOS executor for concolic execution
- Reduced constraints
- Constraint Solver
- Test cases
Remind: Constraints solving is hard!

Constraints

\[
a == 0x1337
\]
\[
\land b * c == 0xc001
\]
\[
\ldots
\]
\[
\land z == 0xc0de
\]

Constraint solver

NP hard

\[\text{Failed} \]
Two types of constraints reduction exist

a == 0x1337
Λ b * c == 0xc001
...
Λ z == 0xc0de

Constraints
Two types of constraints reduction exist

\[
\begin{align*}
& a == 0x1337 \\
& \Lambda b \times c == 0xc001 \\
& \ldots \\
& \Lambda z == 0xc0de
\end{align*}
\]

Constraints

Reduction: Complexity (e.g., Linear reduction)
Two types of constraints reduction exist

Constraints

\[
\begin{align*}
\text{a} &= 0x1337 \\
\Lambda b \cdot c &= 0xc001 \\
\Lambda z &= 0xc0de
\end{align*}
\]

Fast algorithm

Reduction: Complexity (e.g., Linear reduction)
Two types of constraints reduction exist:

- **Fast algorithm**: A fast algorithm with limited expressiveness.
- **Limited expressiveness**: Extended with linear reduction.

Constraints:

\[
\begin{align*}
  a &= 0x1337 \\
  \Lambda b \cdot c &= 0xc001 \\
  \ldots \\
  \Lambda z &= 0xc0de
\end{align*}
\]
Two types of constraints reduction exist

Constraints

\[
\begin{align*}
a &= \text{0x1337} \\
\land b \ast c &= \text{0xc001} \\
\ldots \\
\land z &= \text{0xc0de}
\end{align*}
\]

**Reduction: Complexity** (e.g., Linear reduction)

- \[
\begin{align*}
a &= \text{0x1337} \\
\land 13 \ast c &= \text{0xc001} \\
\ldots \\
\land z &= \text{0xc0de}
\end{align*}
\]

**Reduction: # of constraints** (e.g., Basic block pruning)

- \[
\begin{align*}
a &= \text{0x1337} \\
\land b \ast c &= \text{0xc001} \\
\ldots \\
\land z &= \text{0xc0de}
\end{align*}
\]

**Fast algorithm**

**Limited expressiveness**
Two types of constraints reduction exist:

1. **Complexity Reduction**: (e.g., Linear reduction)
   - Constraints:
     
     \[
     a = 0x1337 \\
     \Lambda 13 \times c = 0xc001 \\
     \ldots \\
     \Lambda z = 0xc0de
     \]

     Fast algorithm

   - Limited expressiveness

2. **# of Constraints Reduction**: (e.g., Basic block pruning)
   - Constraints:
     
     \[
     a = 0x1337 \\
     \Lambda b \times c = 0xc001 \\
     \ldots \\
     \Lambda z = 0xc0de
     \]

     Expressive
Two types of constraints reduction exist

- **Fast algorithm**
  - Limited expressiveness
  - Expressive
  - No algorithmic improvement

**Constraints**

\[
\begin{align*}
a &= 0x1337 \\
\Lambda b \times c &= 0xc001 \\
\ldots \\
\Lambda z &= 0xc0de
\end{align*}
\]

**Reduction: Complexity**
(e.g., Linear reduction)

**Reduction: # of constraints**
(e.g., Basic block pruning)
Two types of constraints reduction exist:

- Reduction: Complexity
  (e.g., Linear reduction)

- Reduction: # of constraints
  (e.g., Basic block pruning)

Never used after a solver can handle non-linear operations.

Constraints:

\[ a = 0x1337 \]
\[ 13 \times c = 0xc001 \]
\[ \ldots \]
\[ z = 0xc0de \]

Fast algorithm

Limited expressiveness

Expressive

No algorithmic improvement

Never used after a solver can handle non-linear operations.
Staged reduction: combine both reduction mechanisms

1. Constraints
   - \( a = 0x1337 \)
   - \( b \cdot c = 0xc001 \)
   - \( z = 0xc0de \)

Reduction: Complexity (e.g., Linear reduction)

Fast algorithm

Limited expressiveness

Expressive

No algorithmic improvement
Staged reduction: combine both reduction mechanisms

1. a == 0x1337
   \( \land 13 \times c == 0xc001 \)
   ...
   \( \land z == 0xc0de \)
   Reduction: Complexity (e.g., Linear reduction)

2. a == 0x1337
   \( \land b \times c == 0xc001 \)
   ...
   \( \land z == 0xc0de \)
   Reduction: # of constraints (e.g., Basic block pruning)

Constraints

- Fast algorithm
- Limited expressiveness
- Expressive
- No algorithmic improvement
Staged reduction: combine both reduction mechanisms

1. \( a = 0x1337 \)
   \( \ \land \ 13 \times c = 0xc001 \)
   ... 
   \( \ \land \ z = 0xc0de \)

   Reduction: Complexity
   (e.g., Linear reduction)

2. \( a = 0x1337 \)
   \( \ \land \ b \times c = 0xc001 \)
   ... 
   \( \ \land \ z = 0xc0de \)

   Reduction: # of constraints
   (e.g., Basic block pruning)

3. \( a = 0x1337 \)
   \( \land b \times c = 0xc001 \)
   ... 
   \( \land z = 0xc0de \)

   Constraints

4. Fast algorithm

5. Limited expressiveness

6. Expressive

7. No algorithmic improvement
Staged reduction outperforms each reduction mechanism

- Setting: Concolic image only, fixed timeout (9 min, 24 hours)
Staged reduction outperforms each reduction mechanism

![Graph showing performance comparison](image)
Staged reduction outperforms each reduction mechanism

Basic block is slow (i.e., expressiveness != efficiency)
Staged reduction outperforms each reduction mechanism

Linear converges too early due to limited expressiveness

Basic block is slow (i.e., expressiveness != efficiency)
Staged reduction outperforms each reduction mechanism

Combining both techniques is useful to achieve higher code coverage!

Basic block is slow (i.e., expressiveness != efficiency)

Linear converges too early due to limited expressiveness
Evaluation

• Effective to discover new bugs in file systems?

• Outperforms the fuzzing-only solution, Hydra?
Hybridra is effective in finding bugs in file systems

• We fuzz for 2 weeks
  • Each fuzzing takes 24 hours
• Target: Linux v5.3 (LKL), but the latest Linux is v5.8

<table>
<thead>
<tr>
<th>File system</th>
<th>File</th>
<th>Function</th>
<th>Type</th>
<th>Concolic</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>btrfs</td>
<td>fs/btrfs/extent_io.c</td>
<td>extent_io_tree_panic</td>
<td>Null pointer dereference</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>fs/btrfs/free-space-cache.c</td>
<td>tree_insert_offset</td>
<td>BUG()</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fs/btrfs/extent-tree.c</td>
<td>btrfs_drop_snapshot</td>
<td>BUG()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fs/btrfs/extent-tree.c</td>
<td>walk_down_proc</td>
<td>BUG()</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fs/btrfs/relocation.c</td>
<td>merge_reloc_root</td>
<td>BUG()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fs/btrfs/root-tree.c</td>
<td>btrfs_find_root</td>
<td>BUG()</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>fs/btrfs/ctree.c</td>
<td>setup_items_for_insert</td>
<td>BUG()</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fs/btrfs/volumes.c</td>
<td>calc_stripe_length</td>
<td>Divide by zero</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>ext4</td>
<td>fs/ext4/super.c</td>
<td>ext4_clear_journal_err</td>
<td>BUG()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f2fs</td>
<td>fs/f2fs/segment.c</td>
<td>f2fs_build_segment_manager</td>
<td>Out-of-bounds read</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hybridra is effective in finding bugs in file systems

• We fuzz for 2 weeks
  • Each fuzzing takes 24 hours
• Target: Linux v5.3 (LKL), but the latest Linux is v5.8

<table>
<thead>
<tr>
<th>File system</th>
<th>File</th>
<th>Function</th>
<th>Type</th>
<th>Concolic</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>btrfs</td>
<td>fs/btrfs/extent_io.c</td>
<td>extent_io_tree_panic</td>
<td>Null pointer dereference</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>fs/btrfs/free-space-cache.c</td>
<td>tree_insert_offset</td>
<td>BUG()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fs/btrfs/extent-tree.c</td>
<td>btrfs_drop_snapshot</td>
<td>BUG()</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>fs/btrfs/extent-tree.c</td>
<td>walk_down_proc</td>
<td>BUG()</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>fs/btrfs/relocation.c</td>
<td>merge_reloc_root</td>
<td>BUG()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fs/btrfs/root-tree.c</td>
<td>btrfs_find_root</td>
<td>BUG()</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>fs/btrfs/ctree.c</td>
<td>setup_items_for_insert</td>
<td>BUG()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fs/btrfs/volumes.c</td>
<td>calcStripe_length</td>
<td>Divide by zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ext4</td>
<td>fs/ext4/super.c</td>
<td>ext4_clear_journal_err</td>
<td>BUG()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f2fs</td>
<td>fs/f2fs/segment.c</td>
<td>f2fs_build_segment_manager</td>
<td>Out-of-bounds read</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hybridra is effective in finding bugs in file systems

- We fuzz for 2 weeks
  - Each fuzzing takes 24 hours
- Target: Linux v5.3 (LKL), but the latest Linux is v5.8

<table>
<thead>
<tr>
<th>File system</th>
<th>File</th>
<th>Function</th>
<th>Type</th>
<th>Concolic</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>btrfs</td>
<td>fs/btrfs/extent_io.c</td>
<td>extent_io_tree_panic</td>
<td>Null pointer dereference</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>fs/btrfs/free-space-cache.c</td>
<td>tree_insert_offset</td>
<td>BUG()</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fs/btrfs/extent-tree.c</td>
<td>btrfs_drop_snapshot</td>
<td>BUG()</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fs/btrfs/extent-tree.c</td>
<td>walk_down_proc</td>
<td>BUG()</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fs/btrfs/relocation.c</td>
<td>merge_reloc_root</td>
<td>BUG()</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>fs/btrfs/root-tree.c</td>
<td>btrfs_find_root</td>
<td>BUG()</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>fs/btrfs/ctree.c</td>
<td>setup_items_for_insert</td>
<td>BUG()</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>fs/btrfs/volumes.c</td>
<td>calc_stripe_len</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ext4</td>
<td>fs/ext4/super.c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f2fs</td>
<td>fs/f2fs/segment.c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Many bugs directly from concolic execution e.g., BUG(x != 0);
Hybridra outperforms the fuzzing-only approach, Hydra

- Setting: Image only (+ Random), fixed timeout (24 hours)
Hybridra outperforms the fuzzing-only approach, Hydra

- Setting: Image only (+ Random), fixed timeout (24 hours)

Concolic execution can help fuzzing in file systems by discovering interesting test cases!
Discussion & Limitation
Discussion & Limitation

• Apply to other applications
  • Our library OS (LKL) also supports network simulation.
  • Thus, it is possible to extend it to network stacks
  • We can apply other user-mode kernel (e.g., Kunit) to test other features

• Limitations
  • Currently, Hybridra does not support floating point and vector operation
  • The limited number of symbols ($2^{30}$) because of shadow memory
    • In our evaluation, this is fine for testing file systems
Conclusion

• Designing a concolic executor tailored for hybrid fuzzing is important for scaling hybrid fuzzing to real-world software
  • Systematic approaches for fast symbolic simulation
  • New heuristics for test case generation

• This dissertation demonstrates this idea with
  • QSYM: Hybrid fuzzing for binary-only applications
  • Hybridra: Hybrid fuzzing for file systems
Acknowledgements

- Georgia Tech
  - Taesoo Kim
  - Meng Xu
  - Jinho Jung
  - Wen Xu
  - Soyeon Park
  - Daehee Jang
- Oregon State University
  - Yeongjin Jang
- Virginia Tech
  - Changwoo Min
- Seoul National University
  - Byoungyoung lee
  - Yunheung Paek
- Microsoft Research
  - Sangho Lee
  - Weidong Cui
  - Xinyang Ge
  - Ben Niu
- University of Pennsylvania
  - Xujie Si
  - Mayur Naik
- UNIST
  - Hyungon Moon
- NSRI
  - Su yong Kim
- KAIST
  - Yongdae Kim
  - Kyoungsoo Park
  - Yung Yi
- University of Michigan
  - Baris Kasikci
  - Upamanyu Sharma
- Arizona State University
  - Ryuou Wang
- Google
  - Chanil Cheon
- Facebook
  - Dhaval Kapil
Thank you!